Chapter 4

Results

In this chapter, we sum up the results of the research of this thesis. Instead, we illustrate the key insights and challenges faced in answering each research question. To obtain our results, we followed the methodology formulated in Chapter 3.

4.1 RQ1. To what extent can we artifically generate program variants for WebAssembly?

As we describe in Section 3.2, our first research question aims to answer how to artifically generate WebAssembly program variants. The main motivation for this research question is that WebAssembly was adopted in 2017, and it lacks of natural diversity [?] . Moreover, compared to the work of Harrand et al. [?] , in WebAssembly, we cannot use preexisting and different programs to provide diversification. In fact, according to the work of Hilbig et al. [?] , the artificial variants created with one of our works contributes to the half of executable and available WebAssembly binaries in the wild.

This section is organized as follows. First we present the general results following *Population size*(Metric 1) for each corpus. Second, we discuss the challenges and limitations in program variants generation. Finally, we illustrate the most common code transformations performed by our approach and answer RQ1.

Program's populations

We summarize the results in Table 4.1. The table is illustrates the Corpus name, the number of functions to diversify, the number of successfully diversified functions (functions with at least one artificially created variant) along with the percentage of successfully diversified functions, the cumulative number of variants taking into account all programs in the corpus and the relation between the increased population size and the original number of functions in the corpus.

We produce at least one unique program variant for 239/303 single function programs for Rosetta with one hour for a timeout. For the rest of the programs (64/303), the timeout is reached before CROW can find any valid variant. In the case of Libsodium and QrCode, we produce variants for 85/869 and 32/1849 functions respectively, with 5 minutes per function as timeout. The rest of the functions resulted in timeout before finding function variants or produce no variants. For all programs in all corpora, we achieve 356/3021 successfully diversified functions, for a 11.78%. As the four and fifth columns show, the number of artifically created variants increased the original population 4.15 times, from 3021 programs to 12547.

Corpus	#Functions	# Diversified	# Variants	Population growing factor
Rosetta	303	**239** (78.88 %)	1906	6.29
Libsodium	869	**85** (9.78 %)	4272	4.92
QrCode	1849	**32** (1.73 %)	6369	3.44
	3021	356 (11.78 %)	12547	4.15

Table 4.1: General program's populations statistics. The table is composed by the name of the corpus, the number of functions, the number of succesfully diversified functions, the number of non-diversified functions and the cumulative number of variants.

Challenges for automatic diversification

We have observed a remarkable difference between the number of successfully diversified functions versus the number of failed-to-diversify functions (third column of Table 4.1). Our approach successfully diversified approx. 79 %, 9.78 % and 1.73 % of the original functions for Rosetta , Libsodium and QrCode respectively.

Setting up the timeout affects the capacity for diversification. A low timeout for exploration gives our approach more power to combine code replacements. We can appreciate this in the last column of the table, where for a lower number of diversified functions, we create, overall, more variants.

Apart from the timeout, we manually analyze programs searching for properties attempting to the generation of program variants using CROW. We identify two main challenges for diversification.

1) Constant inferring We have observed that our approach searches for a constant replacement for more than 45% of the blocks of each function while constant values cannot be inferred. For instance, constant values cannot be inferred for memory load operations because our tool is oblivious to a memory model.

2) Combination computation The overlap between code replacements, is a second factor that limits the number of unique variants. We can generate a high number of variants, but not all replacement combinations are necessarily unique.

Properties for large diversification

We manually analyzed the programs that yield more than 100 variants to study the critical properties of programs producing a high number of variants. This reveals one key factor that favors many unique variants: the presence of bounded loops. In these cases, we synthesize variants for the loops by replacing them with a constant, if the constant inferring [**?**] is successful. Every time a loop constant is inferred, the loop body is replaced by a single instruction. This creates a new, statically different program. The number of variants grows exponentially if the function contains nested loops for which we can successfully infer constants.

A second key factor for synthesizing many variants relates to the presence of arithmetic. The synthesis engine used by our approach, effectively replaces arithmetic instructions with equivalent instructions that lead to the same result. For example, we generate unique variants by replacing multiplications with additions or shift left instructions (Listing 4.1). Also, logical comparisons are replaced, inverting the operation and the operands (Listing 4.2). Besides, our implementation can use overflow and underflow of integers to produce variants (Listing 4.3), using the intrinsics of the underlying computation model.

Listing 4.1: Diversification through arithmetic expression replacement.

```
local.get 0
i32.const 2
i32.mul
```

```
local.get 0
i32.const 1
i32.shl
```

Listing 4.2: Diversification through inversion of comparison operations.

```
local.get 0
i32.const 10
i32.gt_s
```

```
i32.const 11
local.get 0
i32.le_s
```

Listing 4.3: Diversification through overflow of integer operands.

```
i32.const 2
i32.mul
```

```
i32.const 2
i32.mul
i32.const -2147483647
i32.mul
```

4.2 Answer to RQ1.

We can provide diversification for 11.78% of the programs in our corpora. We increase the initial count of programs by a factor of 4.15. We identify the challenges attempting against the automated creation of programs variants, *Constant inferring* and *Combination computation*. Nevertheless, the same constant inferring, complemented with the high presence of arithmetic operations and bounded loops in the original program increased the number of program variants.

4.3 RQ2. To what extent are the generated variants dynamically different?

Our second research question investigates the differences between program variants at runtime. To answer RQ2, we execute each program/variant generated in the

answering of RQ1 for the Rosetta corpus to collect their execution traces and execution times. For each programs' population we compare the stack operation traces (Metric 2) and the execution time distributions (Metric 3) for each program/variant pair.

This section is organized as follows. First, we analyze the programs' populations by comparing the traces for each pair of program/variant with dt_dyn of Metric 2. The pairwise comparison will hint at the results at the population level. We analyze not only the differences of a variant regarding its original program, we also compare the variants against other variants. Second, we do the same pairwise strategy for the execution time distributions Metric 3, performing a Mann-Withney U test for each pair of program/variant times distribution. Finally, we conclude and answer RQ2.

Stack operation traces.

In Figure 4.1 we plot the distribution of all comparisons (in logarithmic scale) of all pairs of program/variant in each programs' population. All compared programs are statically different. Each vertical group of blue dots represents all the pairwise comparison of the traces (Metric 2) for a program of the Rosetta corpus for which we generated variants. Each dot represents a comparison between two programs' traces according to Metric 2. The programs are sorted by their number of variants in descending order.

We have observed that in the majority of the cases, the mean of the comparison values is remarkably large. We analyze the length of the traces, and one reason behind such large values of dt_dyn is that some variants result from constant inferring. For example, if a loop is replaced by a constant, instead of several symbols in the stack operation trace, we observe one. Consequently, the distance between two program traces is significant.

In some cases, we have observed variants that are statically different for which dt_dyn value is zero, *i.e.,* they result in the same stack operation trace. We identified two main reasons behind this phenomenon. First, the code transformation that generates the variant targets a non-executed or dead code. Second, some variants have two different instructions that trigger the same stack operations. For example, the code replacements below illustrate the case.

(1)	`i32.lt_u`	`i32.lt_s`		(3)	`i32.ne`	`i32.lt_u`
(2)	`i32.le_s`	`i32.lt_u`		(4)	`local.get 6`	`local.get 4`

In the four cases, the operators are different (original in gray color and the replacement in green color) leaving the same values for equal operands. The (1) and (2) cases are comparison operations leaving the value 0 or 1 in the stack taking into account the sign of their operands. In the third case, the replacement is less restricted to the original operator, but in both cases, the codes leave the same value

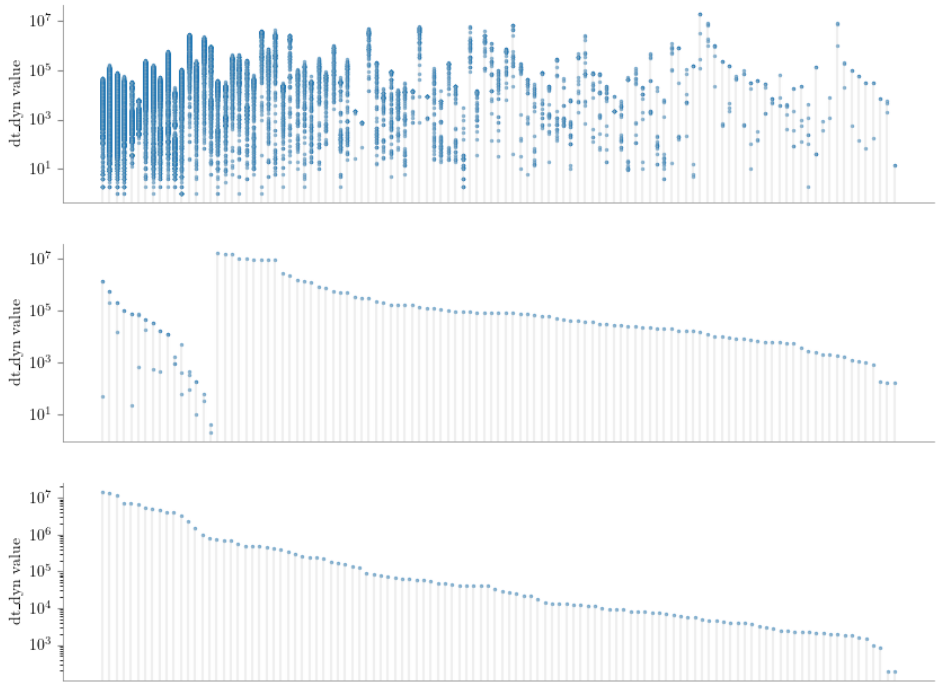

Figure 4.1: Pairwise comparison of programs' population traces in logarithmic scale. Each vertical group of blue dots represents a programs' population. Each dot represents a comparison between two program execution traces according to Metric 2.

in the stack. In the last case, both operands load a value of a local variable in the stack, the index of the local variable is different but the value of the variable that is appended to the trace is the same in both cases.

Execution times.

Even when two programs of the same population offer different execution traces, their execution times can be similar (statistically speaking). In practice, the execution traces of WebAssembly programs are not necessarily accessible, being not the case with the execution time. For example, in our current experimentation we need to use our own instrumentation of the execution engine to collect the stack trace operations while the execution time is naturally accessible in any execution environment. This mentioned reasoning enforces our comparison of the execution times for the generated variants. For each program's population, we compare the execution time distributions, Metric 3, of each pair of program/variant. Overall diversified programs, 169 out of 239 (71%) have at least one variant with a differ-

ent execution time distribution than the original program (P-value < 0.01 in the Mann-Withney test). This result shows that we effectively generate variants that yield significantly different execution times.

By analyzing the data, we observe the following trends. First, if our tool infers control-flows as constants in the original program, the variants execute faster than the original, sometimes by one order of magnitude. On the other hand, if the code is augmented with more instructions, the variants tend to run slower than the original.

In both cases, we generate a variant with a different execution time than the original. Both cases are good from a randomization perspective since this minimizes the certainty a malicious user can have about the program's behavior. Therefore, a deeper analysis of how this phenomenon can be used to enforce security will be discussed in answering RQ3.

To better illustrate the differences between executions times in the variants, we dissect the execution time distributions for one programs' population of Rosetta .

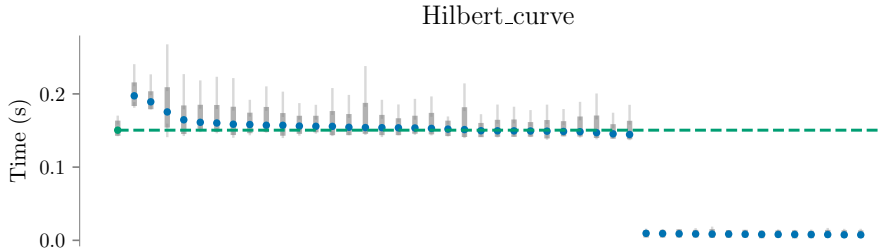

Figure 4.2: Execution time distributions for `Hilber_curve` program and its variants. Baseline execution time mean is highlighted with the magenta horizontal line.

The plots in Figure 4.2 show the execution time distributions for the `Hilbert_curve` program and their variants. We illustrate time diversification with this program because, we generate unique variants with all types of transformations previously discussed in Section 4.1. In the plots along the X-axis, each vertical set of points represents the distribution of 100000 execution times per program/variant. The Y-axis represents the execution time value in milliseconds. The original program is highlighted in green color: the distribution of 10000 execution times is given on the left-most part of the plot, and its median execution time is represented as a horizontal dashed line. The median execution time is represented as a blue dot for each execution time distribution, and the vertical gray lines represent the entire distribution. The bolder gray line represents the 75% interquartile. The program variants are sorted concerning the median execution time in descending order.

For the illustrated program, many diversified variants are optimizations (blue dots below the green bar). The plot is graphically clear, and the last third represents faster variants resulting from code transformations that optimize the original

program. Our tool provides program variants in the whole spectrum of time executions, lower and faster variants than the original program. The developer is in charge of deciding between taking all variants or only the ones providing the same or less execution time for the sake of performance. Nevertheless, this result calls for using this timing spectrum phenomenon to provide binaries with unpredictable execution times by combining variants. The feasibility of this idea will be discussed in Section 4.5.

4.4 Answer to RQ2.

We empirically demonstrate that our approach generates program variants for which execution traces are different. We stress the importance of complementing static and dynamic studies of programs variants. For example, if two programs are statically different, that does not necessarily mean different runtime behavior. There is at least one generated variant for all executed programs that provides a different execution trace. We generate variants that exhibit a significant diversity of execution times. For example, for 169/239 (71%) of the diversified programs, at least one variant has an execution time distribution that is different compared to the execution time distribution of the original program. The result from this study encourages the composition of the variants to provide a more resilient execution.

4.5 RQ3. To what extent the artificial variants exhibit different execution times on Edge-Cloud platforms?

Here we investigate the impact of the composition of program variants into multivariant binaries. To answer this research question, we create multivariant binaries from the program variants generated for the Libsodium and the QrCode corpora. Then, we deploy the multivariant binaries into the Edge and collect their execution times.

Timing side-channels.

We compare the execution time distributions for each program for the original and the multivariant binary. All distributions are measured on 100k executions of the program along all Edge platform nodes. We have observed that the distributions for multivariant binaries have a higher standard deviation of execution time. A statistical comparison between the execution time distributions confirms the significance of this difference (P-value = 0.05 with a Mann-Withney U test). This hints at the fact that the execution time for multivariant binaries is more unpredictable than the time to execute the original binary.

In Figure 4.3, each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary. The dashed line cutting the subplot represents the case in which the two distributions are equal, *i.e.,* for two equal

distribution we would have all blue dots over the dashed line. These plots reveal that the execution times are different and are spread over a more extensive range of values than the original binary. The standard deviation of the execution time values evidences the latter, the original binaries have lower values while the multivariant binaries have higher values up to 100 times the original. Besides, this can be graphically appreciated in the plots when the blue dots cross the reference line from the bottom of the dashed line to the top. This is evidence that execution time is less predictable for multivariant binaries than original ones. This phenomenon is present because the choice of function variants is randomized at each function invocation, and the variants have different execution times due to the code transformations, i.e., some variants execute more instructions than others.

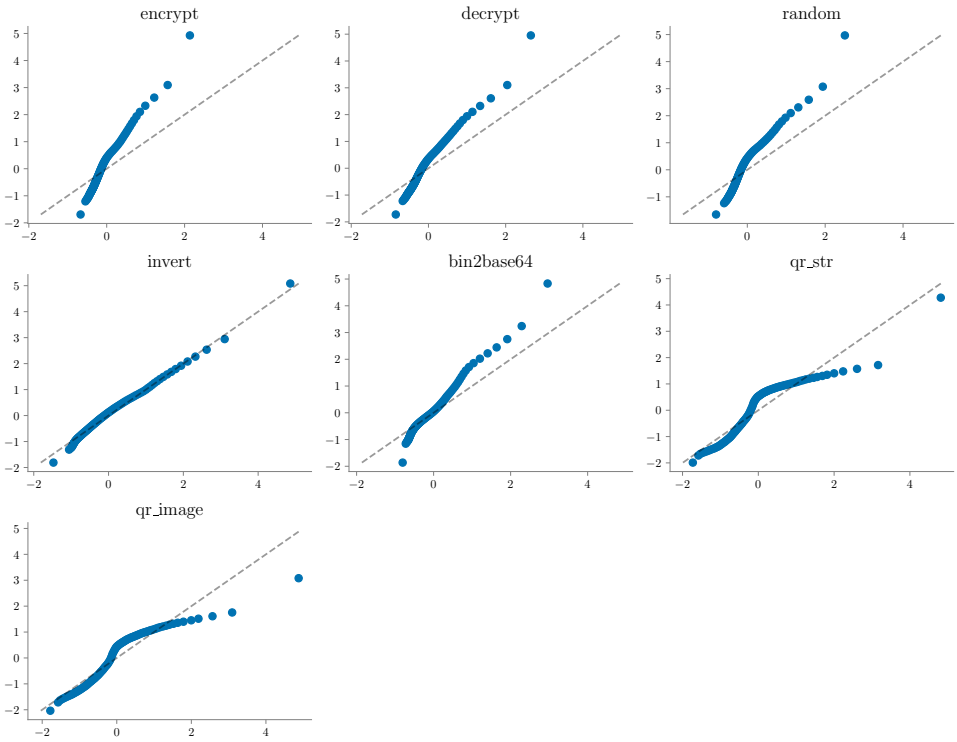

Figure 4.3: Execution time distributions. Each subplot represents the quantile-quantile plot of the two distributions, original and multivariant binary.

4.6 Answer to RQ3.

The execution time distributions are significantly different between the original and the multivariant binary. Furthermore, no specific variant can be inferred from exe-

cution times gathered from the multivariant binary. Consequently, attacks relying on measuring precise execution times [**?**] of a function are made a lot harder to conduct as the distribution for the multivariant binary is different and even more spread than the original one.

4.7 Conclusions

This work proposes and evaluates an approach to generate WebAssembly program variants. Our approach introduces static and dynamic, variants for up to 11.78% of the programs in our three corpora, increasing the original count of programs by 4.15 times. We highlighted the importance of complementing static and dynamic studies for programs diversification. Moreover, combining function variants in multivariant binaries makes virtually impossible to predict which variant is executed for a given query. We empirically demonstrate the feasibility and the application of automatically generating WebAssembly program variants.

Chapter 5

Conclusions